

Ali Abdul-Karim

# Game Development Project with Cocos2D-X

---

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme

Thesis

8 May 2014

## Preface

There is a multitude of reasons as of why I chose to develop a game. I consider myself “hardcore” gamer, I spend most of my past time either playing, making or reading about games and subjects related to them.

Programming, or to be more specific game “modding”, came with Counter-Strike 1.6 back in early 2000s. It all started with me joining a server called something along the lines of “SHMOD Server 20+ heroes”, which got me interested in Superhero mod and programming in general. Secondly the development process of the project at hand was intriguing and a challenging experience because I had never developed anything major in the past with C++.

The game industry in Finland is growing and has already grown quite large, especially after the recent success of not just Angry Birds, but also because of other titles made by smaller companies such as Supercell [1].

Lastly, I’m very interested in working in the field of game development so I thought that this would be a nice introductory card. None the less, this has been a great learning experience in both programming and game design.

Author(s) Title	Ali Abdul-Karim Game Development Project with Cocos2D-X
Number of Pages Date	39 pages 8 May 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engeneering
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer Heini Puuska, Lecturer
<p>This thesis focuses on developing a game project with Cocos2D-X from concept to a finished prototype. This includes but is not limited to game design, UI design, developing with Cocos2D-X and a comparison between Cocos2D-X and other frameworks.</p> <p>The theoretical base consists of online publications, material related to attended lectures and presentations as well as personal discussions with people within the mobile and tablet gaming industry.</p> <p>The result of this thesis is a playable prototype/demo smartphone game project titled” Project Cherry Brawl” for the Android operating system. The art assets of the game were made as part of Linda Karlsson’s bachelor thesis at Novia University of Applied Sciences.</p>	
Keywords	Cocos2D-X, game development, mobile, android

Tekijä(t) Otsikko	Ali Abdul-Karim Cocos2D-X pelikehitysprojektissa
Sivumäärä Aika	39 sivua 8 Toukokuu 2014
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori, Miikka Mäki-Uuro Tuntiopettaja, Heini Puuska
<p>Tässä opinnäytetyössä keskitytään mobiilipeli projektin toteutusta Cocos2D-X pelimoottorin käyttäen. Projektiin kuuluu myös pelin ja käyttöliittymän suunnittelu sekä sen kehitys. Lisäksi vertaillaan Cocos2D-X pelimoottoria muihin mobiililaitteille tarkoitettuja pelimootto-reita.</p> <p>Teoreettinen pohja koostuu online-julkaisuista, luentomateriaaleista, joissa kerrotaan pelien luomisesta ja esitystavasta, sekä henkilökohtaisista keskusteluista älypuhelin- ja tablet-tielialan asiantuntijoiden kanssa.</p> <p>Työn tuloksena on Android-käyttöjärjestelmälle tarkoitettu prototyyppi nimeltä Project Cherry Brawl. Grafiikka tehtiin osana opiskelija Linda Karlssonin opinnäytetyötä Novia ammattikorkeakoulussa.</p>	
Avainsanat	Cocos2D-X, pelikehitys, mobiili, android

## Contents

### List of Abbreviations

1	Introduction	1
2	Cocos2D-X	1
2.1	Architecture	2
2.2	Tools	4
2.2.1	CocoStudio	4
2.3	Comparison	5
3	Project Cherry Brawl	7
3.1	Background Research	7
3.2	Game Concept Design	12
3.3	Scenes	13
3.3.1	Splash Scene	15
3.3.2	Menu Scene	16
3.3.3	Game Scene	17
3.4	Layers	17
3.4.1	HUD Layer	18
3.4.2	D-Pad Layer	20
3.4.3	Game Layer	21
3.5	Actor Sprite	27
3.5.1	Actions	28
3.5.2	Movement	29
3.5.3	Enemies	31
3.5.4	Game AI	31
3.5.5	Collision Detection	33
3.6	Game Performance	35
4	Discussion and Conclusions	36
	References	38

## List of Abbreviations

IDE	Integrated development environment, such as Eclipse or Microsoft Visual Studio.
FPS	a first person shooter is a game which is centered on a projectile type weapon-based combat seen through the first-person perspective.
RPG	a role playing game usually set in a fictional world. Combat system is usually flashy and turn based.
Goto	is a statement that unconditionally transfers control to the statement labeled by the specified identifier.
HUD	or head-up display is the method by which information is visually relayed to the player as part of the games user interface.

## 1 Introduction

The topic of this thesis is a game development project with Cocos2D-X. This thesis covers the whole process from an idea of a game to a finished prototype. This includes an introduction to Cocos2D-X, game design, UI design, AI programming, but is not limited to the aforementioned topics.

The study was conducted in close co-operation with Linda Karlsson; a graphic design student at Novia University of Applied Sciences. She was responsible for the creation of all the games assets, such as drawing/designing the characters, background, enemies, UI, etc. The author of the paper at hand was responsible for the game design and programming. The views and feedback were shared in order to arrive at agreements concerning decisions about the different components of the game, such as the visual look and gameplay.

The thesis begins with explaining the basics of Cocos2D-X and comparing it to other popular frameworks. The thesis then goes into the process of designing Project Cherry Brawl and the background research related to it, after which the thesis covers the different stages of the game development.

## 2 Cocos2D-X

Cocos2D-x [2] is a C++ cross-platform port of the Cocos2D engine for the iOS. It is an open source game engine under the MIT license. With it developers are able to develop not only games but apps and other cross platform GUI interactive programs such as animated backgrounds.

Cocos2D-x allows developers to exploit their existing C++, Lua and Javascript knowledge for developing cross-platform games/applications all the way from prototyping to high performing games/applications saving time and effort.

Whilst Cocos2D-X is not as popular or well known in the west as Unity3D, in Asia it is very popular and can hold its ground to its big brother Cocos2D, and even surpass it because of the cross-platform capabilities [3].

The following sections discuss in detail the architecture of Cocos2D-X, tools compatible with Cocos2D-X and compare Cocos2D-X to other frameworks.

## 2.1 Architecture

The architecture of Cocos2D-X can be separated into three main layers, with sublayers within the main layers. The three main layers consist of the game/application, the components Cocos2D-X provides and the platform its running on respectively as it is illustrated in Figure 1 below.

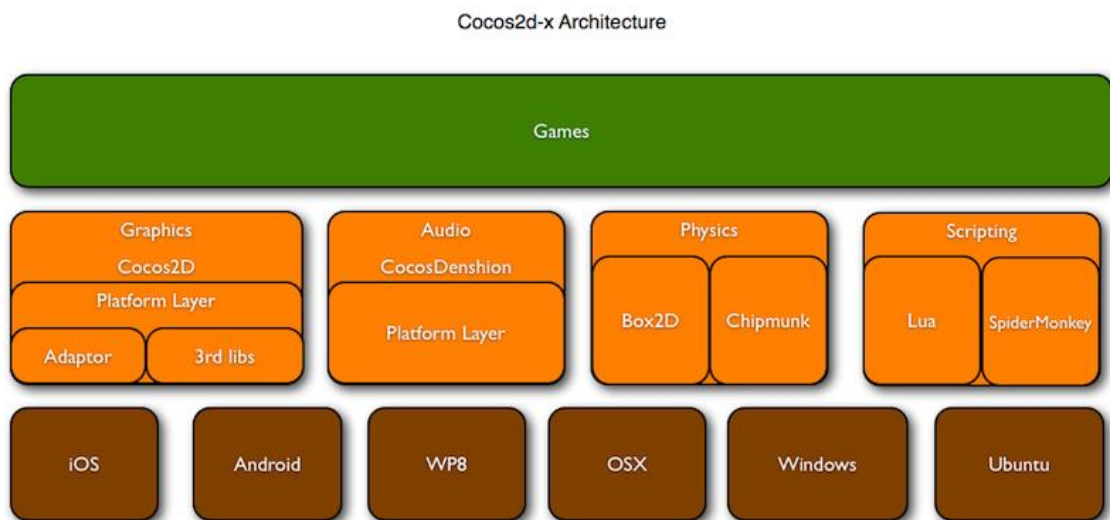


Figure 1. Architecture of Cocos2D-X [4]

The green layer represents the game layer, which will be created by the developers. The orange layer represents all the features that Cocos2D-X provides and lets the developers build their products on and lastly the brown layer represents the platform that the game and Cocos2D-X will be running on. This chapter provides a brief explanation over the most used features of Cocos2D-X.

The director or CCDirector takes care of all the scenes that are created within the project. CCDirector is a shared singleton object, which means there is only one instance of it running at the time. It knows which scene is currently active and handles a stack of scenes to allow various scene calls, such as pausing, putting a scene on hold while another enters, and then returning back to the original scene. So in short, the push, pop, replace and termination of scenes are issued by the director. When one pushes a



new scene onto the stack, the director pauses the previous scene but keeps it in memory. Later when the top scene is popped from the stack, the paused scene resumes from its last state.

CCNode main element in Cocos2D and almost everything in Cocos2D is derived from the CCNode and the most popular ones are CCScene, CCLayer, CCSprite and CCMenu which is covered briefly and then discussed in further detail in Chapter 3. The main features of a CCNode are that it can contain other CCNode(s), it can schedule periodic callbacks (scheduled, unscheduled, etc.) and it can execute actions.

A scene in Cocos2D or a CCScene is an independent piece of the app workflow, basically a “stage”, “level” or a scene of a movie. It is a subclass of CCNode as mentioned before. The game can have as many scenes as chosen, but only one can be active at a time. Figure 2 below illustrates the class diagram of the CCScene.

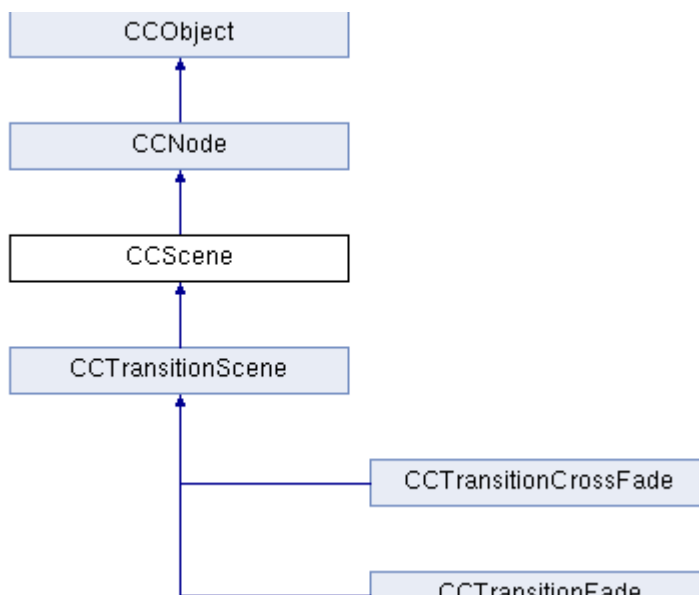


Figure 2. Class diagram of CCScene [5].

A layer in Cocos2D-X is called a CCLayer and it is a CCNode that knows how to handle touch events, draw itself and the children added to it. Layers are used most commonly in defining the appearance and behavior of the game.

A sprite in Cocos2D-X, or commonly referred to as CCSprite, is like any other computer sprite. It is a 2D image that can be moved, rotated, scaled, animated, etc. A CCSprite

can have other sprites as children, which means when the parent is transformed so will the children.

Actions, known as `CCAction` are orders given to a `CCNode` object. These actions can be used to modify the object's attributes like positions, scale, rotation etc. Actions are further elaborate on in Chapter 3.

## 2.2 Tools

This chapter briefly introduces tools that are compatible with Cocos2D-X and that are widely used among the community. These tools provide the user with an easy way to handle different tasks, such as packing textures, creating maps or managing complex data efficiently. These tools include third-party programs.

The Tiled Map Editor is a general purpose tile map editor [6]. It is a free tool that allows easy creation of map layouts. It is very versatile and allows the user to specify more abstract things such as collision areas and object spawn points. It saves this data in TMX format [7].

TexturePacker provides the user with a GUI and a command line tool to create sprite sheets or sprite atlases. It is very versatile and works with a multitude of game engines such as Cocos2D-X and Unity. TexturePacker yields great results while having a lot of adjustability [8].

### 2.2.1 CocoStudio

CocoStudio [9] is the counterpart of CocosBuilder, both of which are game development toolkits. CocoStudio was designed to break down tasks in game development into several roles that wouldn't require the user to know how to program. The roles are as follows, an animation editor for animation and graphic artists, a UI editor for UI graphic artists, a scene editor for game designers and a data editor for game designers. Figure 3 shows how character animation is handled in CocoStudio.

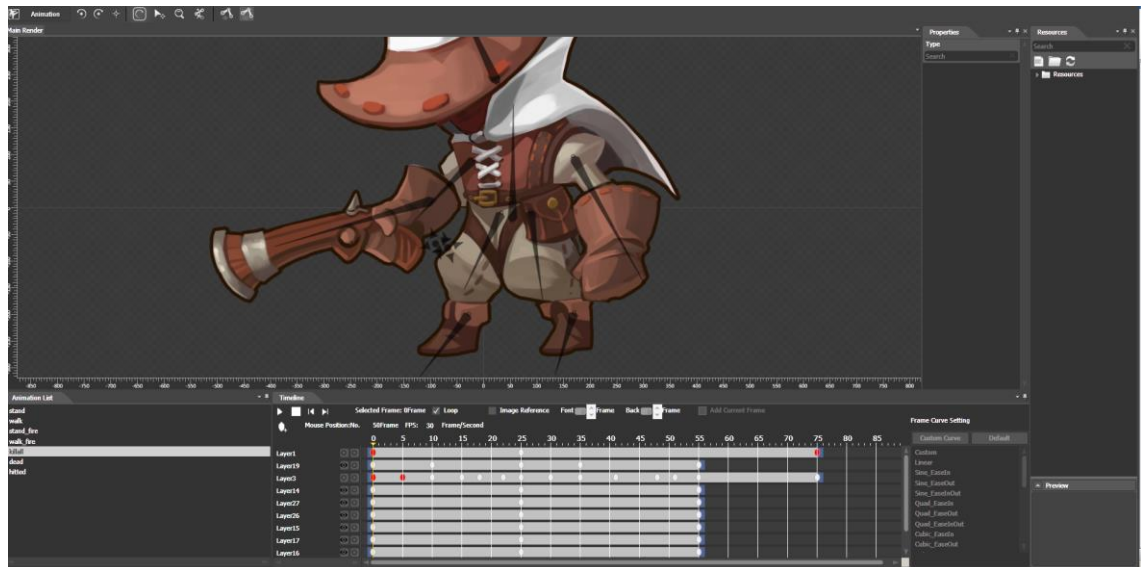


Figure 3. A screenshot of the animation section of CocoStudio

The animation editor brings skeletal animation to Cocos2D-X. Compared to traditional frame animation, skeletal animation provides lower memory consumption, smaller texture size, blending animations together and reusing animations. However skeletal animation has limitations such as it cannot create explosions or an isometric character.

### 2.3 Comparison

This is a short comparison between game engines that fit the criteria of Project Cherry Brawl. The main differences of the game engines are listed below. The following comparison represented in Figure 4 below presents the personal opinions of the author of the paper at hand.

	Price	Orientation	UI	Cross-platform	Language	Community
LibGDX	Free	2D/3D	Very good	Very good	Java	Very good
Unity 3D/2D	Free/Pro 1500\$/ 75\$/month	2D/3D	Basic	Very good	C#, JavaScript, Boo	Very good
Cocos2D-X	Free	2D	Very good	Very good	C++, JavaScript, Lua	Very good

Figure 4. A comparison table of the selected game engines.

LibGDX [10] is a Java game development framework that provides a unified API that works across all supported platforms. The LibGDX framework provides an environment for rapid prototyping and fast iterations.

Unity3D[11] is a complete game development IDE, that comes with a powerful rendering engine fully integrated with a set of intuitive tools for easier creation of 3D/2D content and easy multiplatform publishing. It also comes with its very own Asset Store, where users can download or buy assets created by a very large community.

Unity2D [12] is a new addition to Unity to make it easier for developers to develop 2D games. It comes with a lot of new features to unity that are essential for 2D development, such as a new asset type known as sprite, sprite packing (Pro only), animation window, and many more [13].

Both Cocos2D-X and LibGDX are free and open source, while Unity gives the developer the choice between a free and a pro version. Cocos2D-X is oriented towards to 2D whereas Unity and LibGDX have great 2D support, but focus more on 3D. All three contenders are very well documented and each one of them shows up very high in the stackoverflow tag popularity search, shown in Figure 5.

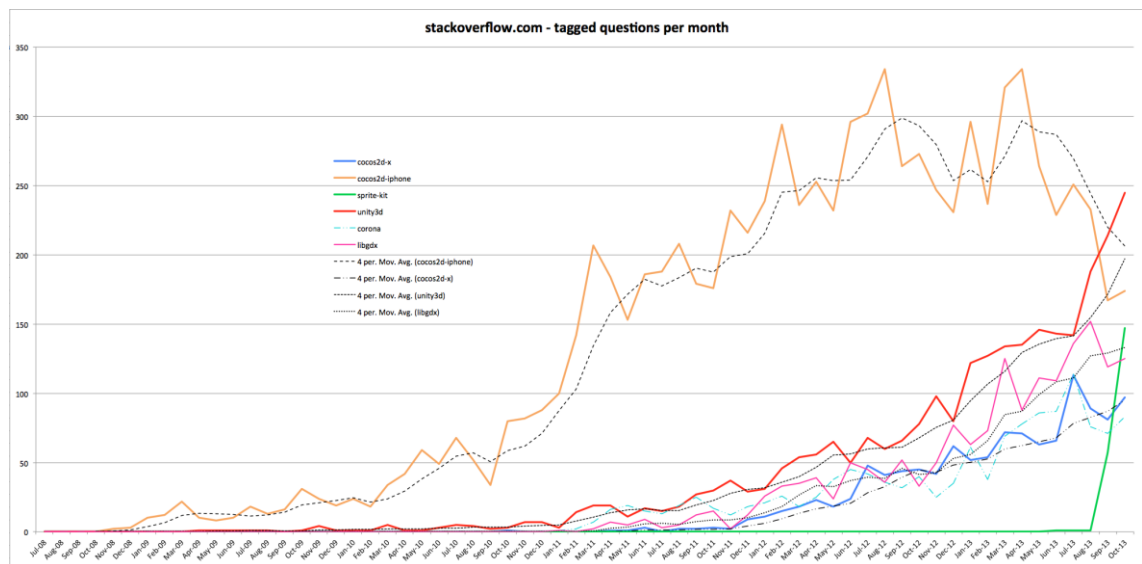


Figure 5. Stackoverflow.com tagged questions per month [14].

While Cocos2D-X and LibGDX provide a very easily accessible and flexible UI implementations, Unity does not and is very limited compared to the previous two. Cocos2D-X, Unity and LibGDX provide superb cross-platform adaptability and have very similar

development cycles. The main programming language for Cocos2D-X is C++, also supporting scripting with Javascript and Lua, while LibGDX works with Java and Unity supports scripting in C#, Javascript and Boo.

Cocos2D-X and LibGDX have a large community behind them and a large arsenal of user generated tutorials and third-party add-ons, while Unity has its Asset Store. These features rapidly decrease development time, and enable graphic artists and game designers to create games from downloadable templates.

### **3 Project Cherry Brawl**

Project Cherry Brawl is a brawler, but may also be referred to as a beat 'em up game. Usually such brawlers feature melee combat between the protagonist, in this case Cherry Lyn, and a large number of underpowered grunt type enemies. The game plays the same way as most brawlers, where the player moves forward (to the right) to fight enemies and defeat them and finally defeat the boss of the stage to complete the level. The game sets the player in the right mindset by elaborating on its backstory:

Cherry Lyn, a girl with extra-sensory powers, ended up staying home from school for a week because of a cold. When Cherry Lyn returned to school things were not quite right. During her absence the science club had developed a computer that could be used as a headset. With the help of the headset the school's students would become more efficient, so the headmaster allowed the production of such headsets. What no one knew was that the science club made some changes to it before it went into production. They added an extra feature to brainwash the students' and therefore take control of the school. Cherry Lyn must now liberate the rest of the school from the science club's control.

#### **3.1 Background Research**

The brawler genre has been around ever since the dawn of arcade machines and old consoles such as the Nintendo Entertainment System. The research that had to be concluded was on how to properly emulate a brawler type game on handheld devices. The research was split into two different categories, the first one being how to properly

make use the mobile devices screen and fit all of the components of the game into it. The second category was how to emulate a controller on mobile devices.

The first step was to try out as many games as possible from the Google Play Store, which were labeled as brawlers or beat 'em up type of games. It was noticed that all the games tested had a couple of key elements had in common. Every game was in landscape mode, to give the player a wider field of view. If the phone would be in portrait mode, it would narrow the player's field of view and would be not very suitable for a brawler type of game.

As shown in Figure 6, most of the mobile brawler games have very cluttered screens. What is meant by that is that when the player has his thumbs on the screen it will reduce the field of view of the player by around 40%. Additionally having other user interface components on the screen further reduces the field of view or just confuses the player. Additionally having buttons cluttered around the sides of the screen will confuse the player as they will blend in with the background.



Figure 6. A screenshot from the game Street Fight [15] from the Google Play Store.

Figure 7 showcases the earlier statement about the reduction of the player's field of view. The view will be hindered by the players own hands and the UI components as well as the background. Further confusion can occur when the background and ene-

mies/UI components blend together, making the player not able to distinguish between them and causing a bad player experience.



Figure 7. A screenshot from the game Street Fight [15] from the play store. View of player.

In games there are different types of UI, often divided into four categories. The four categories are diegetic, non-diegetic, spatial and meta.

An interface that is included in the game world is diegetic, i.e. the player's character can see/hear it. An example could be a holographic picture of the mission commander, who explains the mission for the player's character.

Non-diegetic means that the UI is rendered outside of the game world and is only visible or audible to the player itself and not the character. An example would be Project Cherry Brawl's HUD.

Spatial UI elements are presented in the game world, but don't have to be an entity in the game world. An example would be when the player's cursor hovers over any items or enemies, the item would then get a different outline.

Meta representations can exist in the game world, but aren't categorized as spatial. An example of a meta representation can be blood splatter on the player's camera to indicate damage. Figure 8 reflects my previous statements.

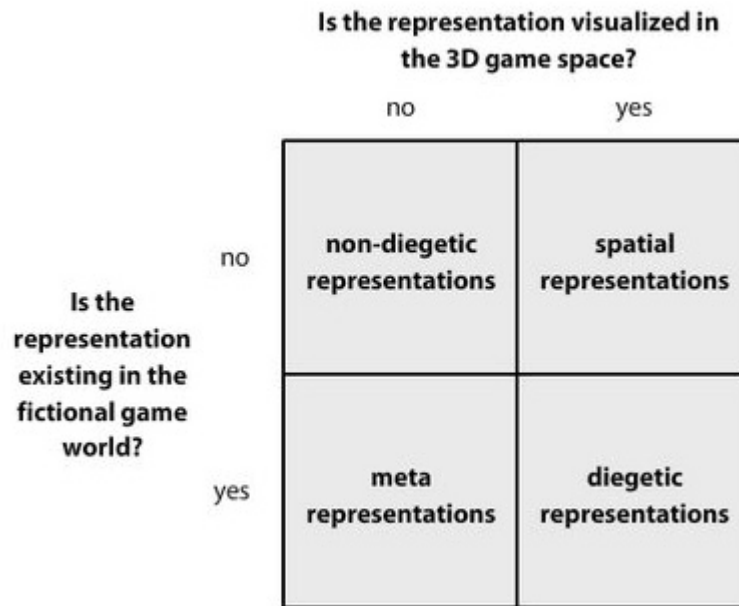


Figure 8. Four categories of UI design [16].

Project Cherry Brawl UI belongs mostly to the non-diegetic category because the UI is outside the game world and the characters are unaware that it exists. There was one exception and that is the numbers that appear above the characters when they take damage. These numbers belong to the category of spatial interfaces as they appear in the game world while the characters are still unaware of the existence of the numbers [16].

After creating and testing a prototype where the UI components of Project Cherry Brawl were displayed in the same fashion as Street Fight, the verdict was to scrap the idea because the player himself will be blocking both ends of the screen with his hands, depending on the situation.

The idea for the current UI layout came from wanting to display everything to the player as clear as possible. After having a meeting with Linda Karlsson it was concluded that the UI is as can be seen in Figure 9. With this design the player's information shows right in front of him, while having his hands on each side of the screen. This not only keeps the screen from being cluttered, but also gives the player more of a retro feel of using an actual controller. Additionally none of the UI elements will appear on the actual gameplay and the player will in no way block his own view as presented in Figure 9 and Figure 10.



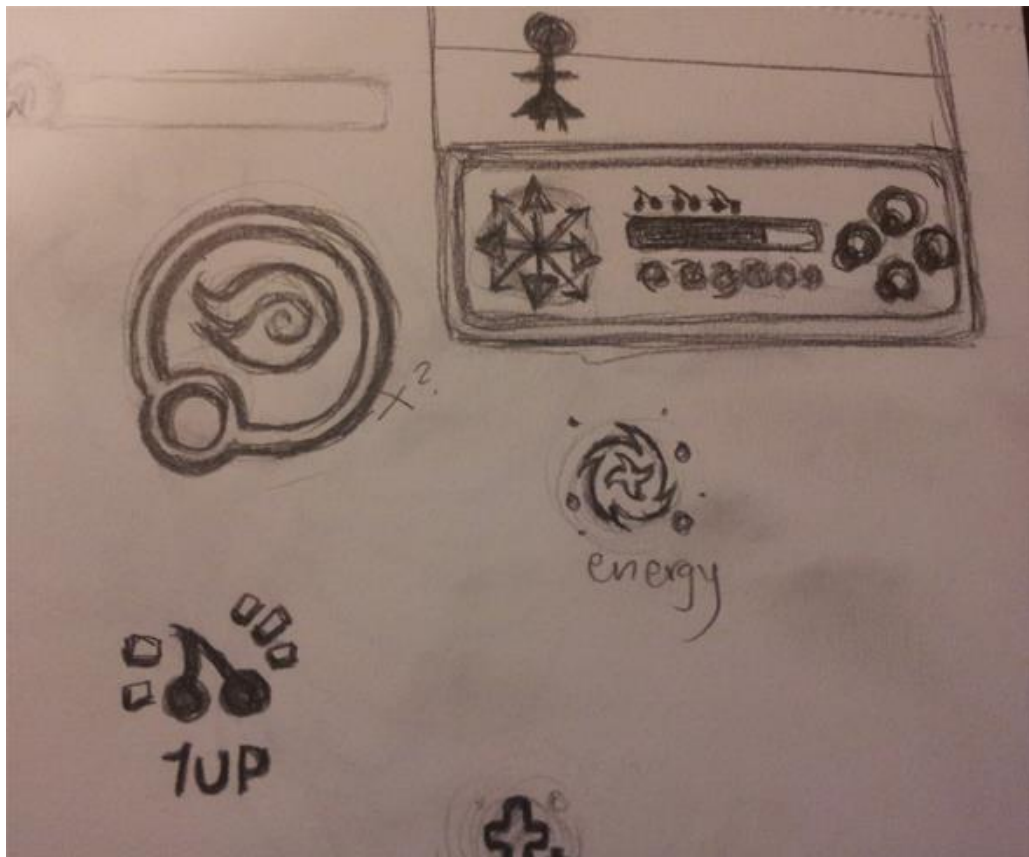


Figure 9. UI prototype for Project Cherry Brawl.

With the current design, the player loses little of room of movement, but in exchange the games characters and map were slightly scaled down. This way the player has more room to play with, while keeping the screen tidy as demonstrated in Figure 10.

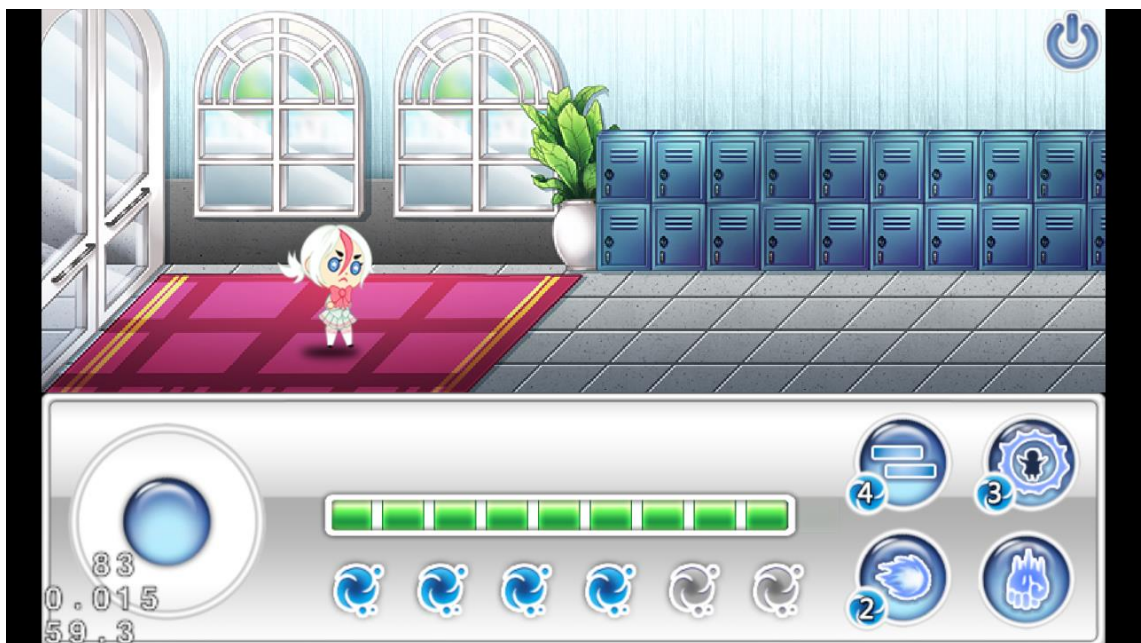


Figure 10. A screenshot from Project Cherry Brawl, demonstration of the finished UI.

The next issue arose when the games assets had to be created. Usually mobile games have to provide different assets to support a multitude of aspect ratios. That way the same game can be enjoyed on popular mobile devices such as tablets and phones. Because of limited resources and time the project had to focus on a certain type of mobile devices. The chosen aspect ratio was 15:9 which made the game resolution 480x800 pixels. This way the device would add black borders similar to above if the devices resolution is bigger than the chosen 480x800pixels. This was implemented with multi-resolution support of Cocos2D-X [17].

### 3.2 Game Concept Design

Project Cherry Brawl had a lot of different concept designs and took inspirations from a multitude of great titles such as Double Dragon [18], Streets of Rage [19], Final Fight [20] and Teenage Mutant Ninja Turtles: Turtles In Time [21].

Having attended IGDA [22] on March 2014 [23], there was a guest lecture by an employee from DeNA [24]. There she explained the differences between western and eastern mobile games. So called Asian mobile games are very flashy and give a lot of feedback to the player. The feedback was most of the time positive, very flashy and shown in with vibrant colors. That way even the smallest step the player accomplishes in the game feels like an accomplishment. She furthermore explained how that is directly inherited from Asian culture. With her statement in mind Cherry Lyn's signature attack was created as can be seen in Figure 11.

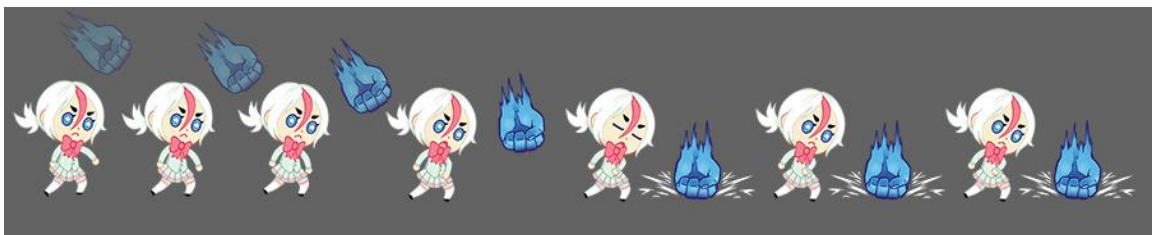


Figure 11. Prototype of first attack, game Project Cherry Brawl.

Having Cherry Lyn's attacks be flashy proved to be a nice addition to the game. In most brawlers the combat is only hand-to-hand, but having decided to add different

attacks, such as a big circular energy wave and an energy projectile as can be seen in Figure 12, gave more of an RPG element to the game.

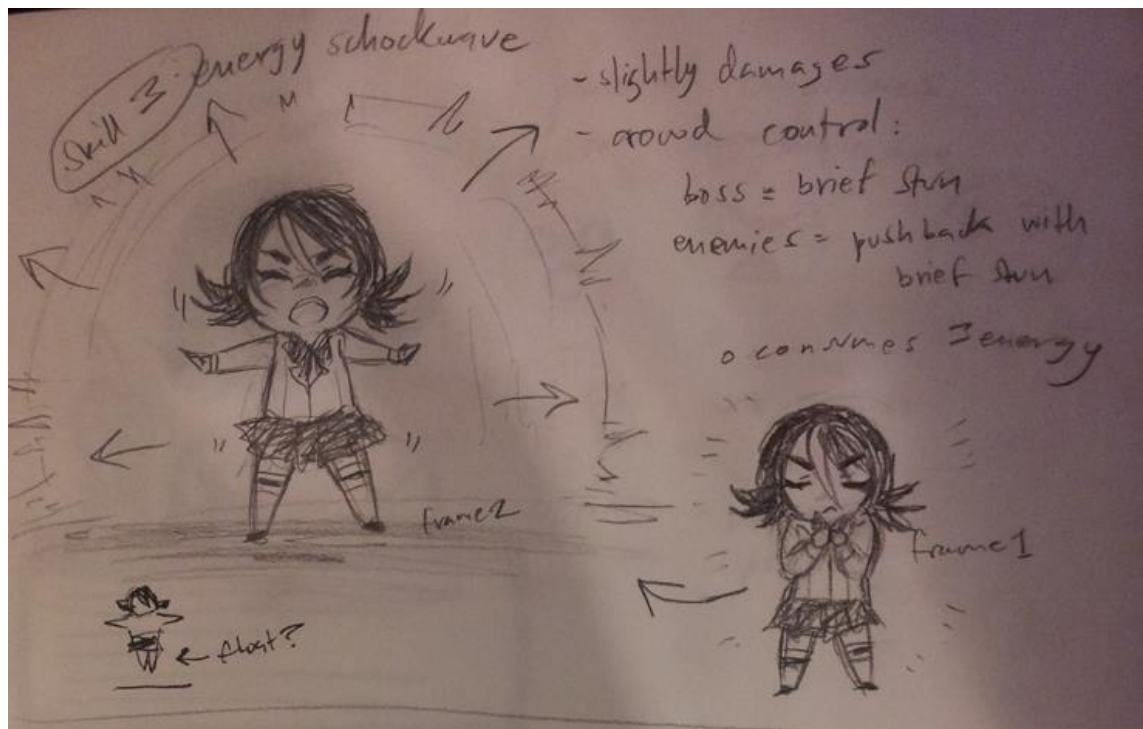


Figure 12. Prototype of the 3<sup>rd</sup> attack of Cherry Lyn, Project Cherry Brawl.

Figure 12 showcases early concept of the aforementioned circular energy attack. As can be noted, the characters attacks draw heavy inspiration from RPG games. It was intentional that the game would not include any physical contact.

### 3.3 Scenes

This chapter further elaborates on the previous statement about scenes and how they were used in Project Cherry Brawl. It is a good practice to use a CScene as a parent for all the games different CCNode(s), such as layers.

In Project Cherry Brawl there are three different scenes, which are the splash scene, menu scene and the game scene. They are used to contain different layers and other CCNodes inside and order them. The creation process of each of the aforementioned scenes will be discussed shortly in the following chapters.

The difference between the init methods in Cocos2D-x and Cocos2D are that in Cocos2D-X the init method returns a Boolean, while in Cocos2D it returns an id. Now what this does is that it promotes defensive programming. Previously, the developer of Cocos2D-X [25] wrote code that managed its clean up with goto's:

```
#define check(ret)  if(!ret) goto cleanup;

void func()
{
    bool bRet = false;

    bRet = doSomething();
    check(bRet);
    bRet = doSomethingElse();
    check(bRet);

    bRet = true;

cleanup:
    // Do clean up here

    return bRet;
}
```

Snippet 1. First version of how inits were handled.

As can be noticed by the code above, if anything goes wrong along the way it would jump to the clean-up at the end of the function. Each call to a function returns whether it was successful or not. Then the check macro is used to see if bRet is true and if it is not it jumps straight to the cleanup label. Technically speaking, there is nothing wrong with this, but they met a problem because scripting languages do not have a goto instruction. That's why it was changed to:

```
#define CC_BREAK_IF(cond)  if(!cond) break;

void func()
{
    bool bRet = false;

    do {
        bRet = doSomething();
        CC_BREAK_IF (bRet);
        bRet = doSomethingElse();
        CC_BREAK_IF (bRet);

        bRet = true;
    } while (0);

    // Do clean up here

    return bRet;
}
```

Snippet 2. Cross-Platform version of how initializations are handled.

This has the exact same effect as previously, but uses break as a goto mechanism to jump to the code after the do while loop. It is generally a good practice intended to ensure the continuing function of a piece of software in spite of unforeseeable usage of said software. This was created because of Cocos2D-HTML5/JS [26].

This might have gone slightly off topic of scenes, but it was an essential part of scenes and had to be elaborated. To keep unforeseen errors and bugs from appearing every CCNode should be initialized in the aforementioned way.

### 3.3.1 Splash Scene

A splash scene is what one sees most of the time when opening any program/game. It shows credits, company name, etc. for example when booting up windows, it shows the windows greeting screen, which leads to the login screen.

The splash screen was the first building block of Project Cherry Brawl, as it served as a hello world. All that should happen in this scene is that it would load up a logo, display it and after a certain amount of time call director and change scenes to the menu scene.

This was done in the following order. After initializing the Splash layer it then proceeded to create a CCScene and add the aforementioned Splash layer to it as a child. After the Splash layer had been added, calling the CCDirector to change scenes was not an optimal solution as that would change scenes instantly. The desired result was to have the Splash screen linger long enough for the player to read the logo and then change the scene, as can be seen in Snippet 3.

```
CCCallFunc* changeScene = CCCallFunc::create (this, call_func_selector
(DisplayScene));

CCDelayTime* delayAction = CCDelayTime::create (2.0f);
this->runAction (CCSequence::create(delayAction, changeScene, NULL));
```



Snippet 3. Change scene action in the splash scene of Project Cherry Brawl.

Additionally by giving the Splash layer a CCAction, which can be given to all CCNode objects, the desired effect was achieved using the delayAction, causing the action to run after a delay of two seconds.

### 3.3.2 Menu Scene

The menu scene for this game is very straight forward, unlike other games that have complex menu systems. The menu scene has two buttons which would either display the controls of Project Cherry Brawl or change scenes to the game scene. Creating the Menu scene the same way as the Splash scene proved problematic. The problem was that some of the child CCNodes such as particle effects and the logo were overlapping in an unwanted way.

Figure 13 showcases the finished menu scene, demonstrating the ordering of child nodes.



Figure 13. Project Cherry Brawl main menu screen, demonstration of different Z values for different elements of the scene.

The problem was solved by adding all of the menu's components as child nodes to the Menu scene. That way the display order of the children could be reordered. The higher the value the more the specific node is above other nodes.

### 3.3.3 Game Scene

This is the main scene where the whole game plays out. The creation is similar to the previous two scenes. Unlike the previous splash and menu screens, the game scene was to have a lot of child nodes in it, and managing that with one layer was not an option. In the long run it would cause the code to become very messy, hard to understand and most importantly it will prove very difficult to maintain.

Thus the need to create separate layers to contain more game objects arose, such as HUD objects or game objects. The creation of a HUD layer and a Game layer were an essential part of the game. The HUD layer would handle everything related to the UI while the Game layer maintains everything related to the actual game world. Both layers are further elaborated on in the next chapter.

## 3.4 Layers

CCLayers are essential to keep the code clear and easily comprehensible while defining your games appearance. Layers are subclasses of the CCNode class, so they may contain images, animations, labels, button, sprites, etc. The defining factor about layers is that they implement the TouchEventsDelegate protocol, which means that the player can interact with the layer with touch events, such as tapping, swiping, holding and moving.

Project Cherry Brawl makes use of the implementation of three different layers, each one representing a component of the game as can be seen in Figure 14.

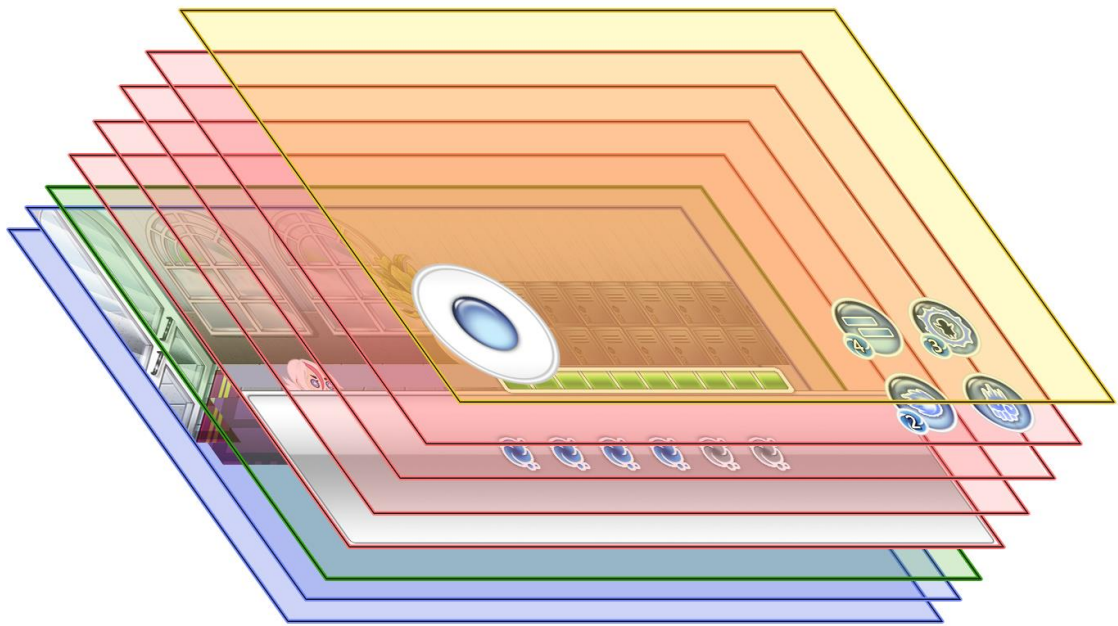


Figure 14. Layer sample of Project Cherry Brawl.

The finished build of HUD, game and D-Pad layers and how they are placed can be seen above. Additionally it shows how even with two main layers one can have a lot of visual depth in a game. Blue and green layers represent the game layer, red represents the HUD layer and yellow represents the D-Pad layer, but more on those in the following chapters.

### 3.4.1 HUD Layer

When designing the HUD layout the following factors had to be taken into consideration. Easily accessible buttons for the players skills, for the go back to menu scene button and the tap to continue button for the dialogue section. Icons for the players and the enemies name tag, and both of their portrait textures. Handling the dialogue state of the game dynamically, meaning to hide and show the specific child nodes of the HUD layer. The players health bar and energy bar, and handling its update.

Because the HUD is heavily dependent on the game layer, the solution was to create the child nodes in the HUD layer and initialize them depending on their dependencies. Initializing the user's skill buttons in the HUD layer would have made me create a reference of the Game layer object and pass it to the HUD layer. In doing that the code would become very hard to interpret and it would be very messy.



Objects such as the dialogue box, the background of the hit points, the border of the hit points, game dialogue and name tags as well as the D-Pad layer were initialized in the HUD layer. Whereas the exit button, player buttons, player health and energy bars were initialized in the game layer and added to the HUD layer as children. That way only one reference of the HUD layer object had to be passed to the game layer. This provided easy access to all the HUD layer components from the Game Layer. Figure 15 showcases the order of the hit point's background, the hit points and the hit point's border respectively.

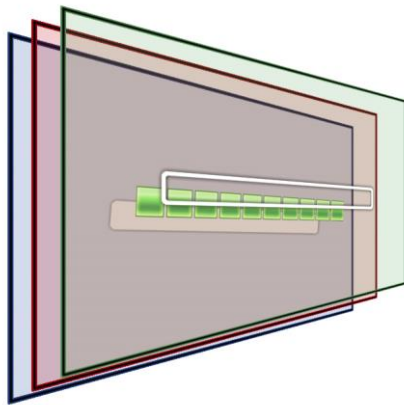


Figure 15. The order of the hit point's bar in Project Cherry Brawl. Showing the hit point's background, the hit points and the hit point's border respectively.

The dialogue state is also handled in the HUD layer. By creating two functions called `dialogueModeOn` and `dialogueModeOff` the child nodes could be dynamically set invisible and back to visible. The players and enemies dialogue pictures and name tags are also handled in the HUD layer. When calling `dialogueModeOn` one could call whom he or she would like see to talk, for example to have the main character talk, one would call

```
_hud->cherryTalks (true, 1); // frames from 1-4
```

Snippet 4.

The parameters are for setting the child visible and the picture frame respectively, a similar function was made for the enemy as well.

### 3.4.2 D-Pad Layer

A D-Pad [27], also known as a direction pad or control pad, is a thumb-operated four-way directional control where each direction is a button, and is found in most of the game console controllers. The game D-Pad is a very important component of game play, if not the most important component. Having an unresponsive D-Pad would cause players to have bad experiences with the game.

In the earlier iterations of the D-Pad, it was a 4-way oriented D-Pad. That proved to be very unresponsive, because especially in brawlers the player has to be able to move in all eight directions. The final iteration of the D-Pad can be seen in Figure 16.



Figure 16. Game D-Pad object outlines. In the middle is the button sprite.

As can be noticed, the image provided shows two different outlines. The reason behind this is the human thumb. After thorough testing of the D-Pad the following kept happening:

As the player would try to move diagonally, the button sprite would snap back to the middle. This was caused by the size of the players thumb and how the touch screen would interpret where the player is pressing, when the players thumb would move it to the edge of the circle.

By making the active touch area of the D-Pad rectangular, it made sure that the control the player has over the character would be very responsive. Additionally it made sure that the button sprite would not suddenly snap back while the player was moving the

character diagonally. Furthermore to keep the feeling of a D-Pad, the button sprite was given a circular boundary to enhance the feeling of using a D-Pad.

### 3.4.3 Game Layer

Designing the game layer was one of the more challenging parts of this project. Even though everything is played inside the game scene, the game layer had to take care of all the logic of the game, the character movement, the tile map and the game effects. The following features had to be present in the layer:

The loop of the game and initialization for all the game objects and some HUD layer objects. It has to take care of loading the tile map and manage the cut scenes and dialogue. Display and update all the child nodes, such as the player's character, enemies and HUD layer components such as the player's health. Handle the players input on the D-Pad, reorder all the child actor sprites positions and update the players view point. So the game layer can be considered the core of this game. It holds everything together and deals interactions between objects.

Fortunately the CCLayer node supports the `scheduleUpdate`, which adds a scheduled update to the layer. By overriding the `update (deltaTime)` function which it will provide, one can create a game loop. A game loop [28] is the central component of a game from a programming standpoint. It allows the game to keep running, regardless of there being any user input or not.

Using TexturePacker helped keep all of Project Cherry Brawl's assets in order and reduce the space that they would use. As can be seen in Figure 17, even such a small game prototype can have a lot of textures and assets in general. By compressing all the assets into one picture the usage of the batch node was made available.

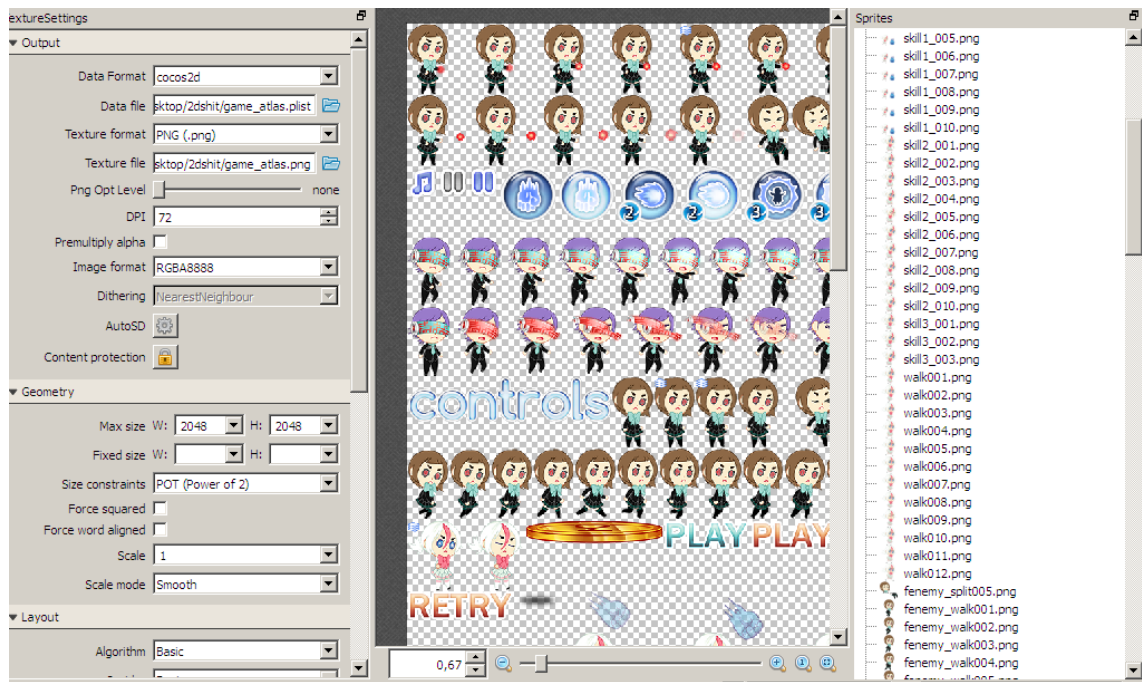


Figure 17. Sprite sheet created by TexturePacker of Project Cherry Brawl's assets.

Having added all the assets into the TexturePacker, it created a sprite sheet. This not only sped up the game's loading time of the textures, but it was also an essential part of using the `CCSpriteBatchNode` as once it loaded the assets they would be available in the shared sprite frame cache.

In an earlier iteration of Project Cherry Brawl the game objects such as the player character, enemies, etc. were added to the game layer as children. That made the code very hard to maintain and update. Additionally the game would perform poorly, even when there were only ten enemies on the screen at a time.

What the batch node provides is that if it contains child nodes, it will draw them all in one single OpenGL call. That is often referred to as a "batch draw". In the absence of a batch node OpenGL draw calls will be called as many times as number of the children, in this case sprites such as the player character and enemies.

A `CCSpriteBatchNode` can reference only one texture, image file, texture atlas or in this case a sprite sheet. Adding all the actor sprites to a `CCSpriteBatchNode` greatly increased the performance of the game, giving the player the desired frames per second even under the heaviest possible stress the game could provide to the phone.

Reordering the sprites at first very hard to grasp at first. In one of the early iterations the sprites had set z values which looked very unrealistic as can be seen in Figure 18.



Figure 18. Project Cherry Brawl, without z sorting. Red circles indicate errors.

Another iteration of this problem setting each actor sprites z value depending on their position at the time of creation, but that yielded the same unwanted result. Because every single actor sprite is a child node of the CCSpriteBatchNode, the dynamic reordering of the child nodes was possible as can be seen in the Snippet 5. The result can be seen in Figure 19.

```
CCObject *pObject = NULL;
CCARRAY_FOREACH (_actorsAtlas->getChildren (), pObject)
{
    ActorSprite *sprite = (ActorSprite*) pObject;
    _actorsAtlas->reorderChild (sprite, (_tileMap->getMapSize
    ().height * _tileMap->getTileSize ().height) - sprite->getPosition
    ().y);
}
```

### Snippet 5. Reorder child function, taken from Project Cherry Brawl



Figure 19. Project Cherry Brawl, enemy/shadows sprites rearrange.

Since version 0.8.1 Cocos2D-X provides the CCTMXTiledMap which is a CCNode that knows how to parse and render a TMX map. While rendering each tile, they will be created as CCSprites. This means that each tile can be rotated, moved, scaled, tinted, etc. at will. Since the tile map is created from a tileset image, it will be loaded into its own CCTextureCache, which speeds loading times and performance. Each layer of the tilemap will be created using CCTMXLayer a subclass of CCSpriteBatchNode. By creating the tile map in Tiled, the parsing of the map was easily achieved using the CCTMXTiledMap node.

The game map was created in Tiled using two layers, the wall and the floor layer. Just like layers in Cocos2D-X the layers in tiled provided the user the ability to draw objects on top of objects to give the map a feeling of depth as can be seen in Figure 20.



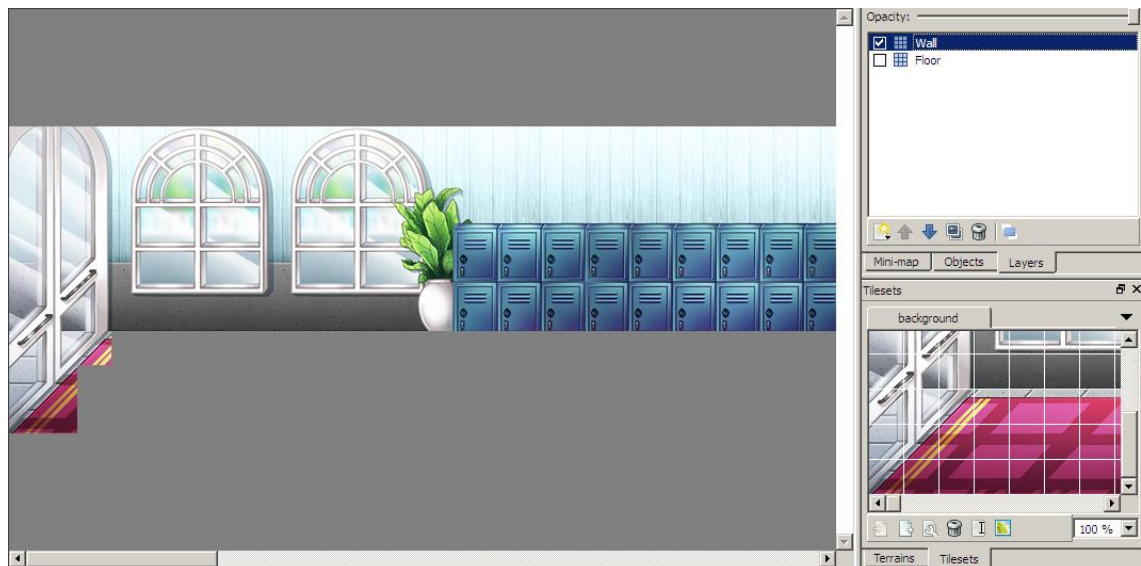


Figure 20. Demonstrating layers in Tiled, wall layer. Taken from Project Cherry Brawl.

As demonstrated in Figure 20 all of the wall tiles were drawn on the wall layer. In Figure 21 the floor component of the game was created. Because this layer is under the wall layer, anything added into the floor layer would go under the wall layer.

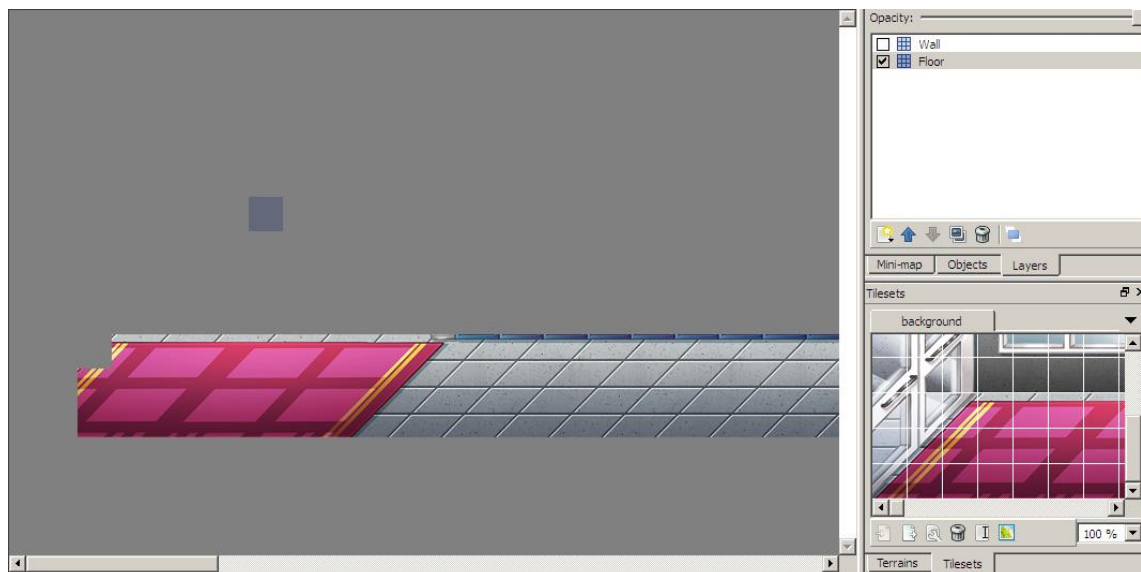


Figure 21. Demonstrating layers in Tiled, floor layer. Taken from Project Cherry brawl.

While planning Project Cherry Brawl the intention was for it to be a prototype. It would demonstrate the core mechanics of the game. The prototype was designed to take roughly one to three minutes of gameplay to complete. Late into the project the game's atmosphere seemed rather dull. There was no incentive for Cherry Lyn to fight, thus

arose the need of creating the aforementioned backstory for the game. The game's story was to be delivered through small cut scenes between fights or check points in the game.

Having decided to add the cut scenes, two states were created for the game. The dialogue state meant that the enemies, player and boss would have scripted movements, rather than the players input. The running state meant that the enemies would move according to their AI and the character would move based on the players input.

Handling the different cut scenes proved to be tricky, but with the help of checkpoints when the game is in dialogue state, the cut scenes would seamlessly transition from one to another. Figure 22 showcases the transitions of the game's dialogue between Cherry Lyn and an enemy grunt.



Figure 22. Project Cherry Brawl, transitions from dialogue state to gameplay state back and forth.

Setting the center of the viewpoint proved to be a problem of its own. Trying out different approaches to this problem, the most optimal way seemed to be the following:

By determining the player characters current position and comparing it to the width of the screen divided by 2 with the MAX macro, which returns the greater value between the two aforementioned variables. One would use the aforementioned value and compare it to the tile maps width minus the screens width divided by two using the MIN macro, which returns the smaller value of the two variables. The same procedure is done to the height of the viewpoint. Using the ccpSub macro which would subtract the center of the screen and the previously calculated point in the variable x as demonstrated in Snippet 6.

```
int x = MAX(position.x, SCREEN.width / 2);
int y = MAX(position.y, SCREEN.height / 2);
x = MIN(x, (_tileMap->getMapSize ().width * _tileMap->getTileSize
().width) - SCREEN.width / 2);
y = MIN(y, (_tileMap->getMapSize ().height * _tileMap->getTileSize
().height) - SCREEN.height / 2);
```



```
CCPoint actualPosition = ccp(x, y);

CCPoint centerOfView = ccp (SCREEN.width / 2, SCREEN.height / 2);
CCPoint viewPoint = ccpSub (centerOfView, actualPosition);
this->setPosition(viewPoint);
```

Snippet 6. Camera position update.

This way, the viewpoint of the game does not go under the HUD layout and the camera would follow the player.

### 3.5 Actor Sprite

When designing the game, and especially the player's character and enemies, it became clear that the CCSprite node alone did not provide all the needed functionality. Additionally the creation of an abstract that provides a simple interface for creating the enemies and the player's character was needed. This way there was no need to rewrite basic functionality when creating new enemy types.

The actor sprite class has to handle the abstract creation of all possible animations such as the player standing still or walking. The creation of the bounding box for the actor sprite as well as different bounding boxes for attacks additionally the actor sprite has to handle all related box transformations. Very generically programmed attacks and scripted movements were created to easily be able to modify, add or remove in the future. The actor sprite has to be aware on where it is looking and handle its own movement. Furthermore the actor sprite has to keep track of the state that it is in as well as various variables, such as its own hit points, energy, and movement speed, etc. The actor class also inherits the CCSprite node, which provides it all of the CCSprites functionality.

While designing the actor sprite class, a decision was made that every actor sprite object has the possibility to be only in one state at a time as can be seen in Snippet 7.

```
typedef enum _ActionState {
    kActionStateNone = 0,
    kActionStateIdle,
    kActionStateAttack,
    kActionStateWalk,
    kActionStateHurt,
    kActionStateKnockedOut,
    kActionStateDialogueWalk
```

```
} ActionState;
```

Snippet 7. States of the ActionSprite class.

This means that the sprite can attack, stand idle, move or be knocked out, but cannot do two things at the same time.

### 3.5.1 Actions

A CCNode such as CCSprite will store information about the position, rotation, visibility and opacity of itself. In Cocos2D-x, there is a CCAction class to change each of these values over time, performing various transformations or actions.

Another criterion for the actor sprite was to create such actions for the sprite such as attacking, standing idle or walking. The implementation of these actions was possible with the addition of the CCAction object. By creating the various actions in the actor sprite class, and initializing them in the classes that inherit it. The author was able to easily maintain the generic actions that every actor sprite would require, such as idle, moving, attack and getting knocked out as well as adding new ones with ease. Cocos2D-X has an incredibly flexible system that allowed the creation and combination of different actions and transformations to achieve the desired results. Snippet 8 showcases how the idle animation for the player is created.

```
//idle animation
CCArray *idleFrames = CCArray::createWithCapacity (12);
for (i = 1; i < 13; i++)
{
    CCSpriteFrame *frame = CCSpriteFrame-Cache::sharedSpriteFrameCache()-
>spriteFrameByName(CCString::createWithFormat("idle%03d.png", i)-
>getCString());
    idleFrames->addObject(frame);
}
CCAnimation *idleAnimation = CCAnimation::createWithSpriteFrames(idleFrames, float(1.0 / 12.0));
this->setIdleAction(CCRepeatForever::create(CCAanimate::create(idleAnimation)));
```

Snippet 8. Creating the idle action for the player character.

All of the different animations for the actor sprites were created in the same manner.

### 3.5.2 Movement

The movement and bounding box transformations were troublesome, mostly because the player and enemies should be able to attack in all directions. Additionally getting the enemies to look in the right direction, in this case towards the player, was a challenge.

Changing the direction of movement for the the player's character was determined by where the player would move the D-Pad to. By checking if the velocity.x value is negative or positive, the game would change the player's direction accordingly. The velocity value is calculated in Snippet 9.

```
_velocity = ccp (direction.x * _walkSpeed, direction.y * _walkSpeed);
if (_velocity.x >= 0)
{
    this->setScaleX(1.0);
}
else
{
    this->setScaleX(-1.0);
}
```

Snippet 9. Determening the players position based on speed and current position.

In Snippet 9 setScale is for setting where the sprite in question is currently looking at, it being positive means that it is looking to the right and it being negative means that it is looking to the left. Because enemies are not controlled by the player, setting their view point was determined by where the player is standing and that would determine if the desired movement would be to the left or to the right.

The bounding boxes for each actor sprite were created by setting an origin point and creating a box out of it. The origin point is in the bottom left corner of the sprite as demonstrated in Figure 23.



Figure 23. Cherry Lyn's hit box from the game Project Cherry Brawl.

To update the bounding boxes a `transformBoxes` function was created. The `transformBoxes` function would handle bounding box transformation every time the action sprites update function would be called. The bounding boxes are essential for collision detection, but that will be discussed in the collision detection chapter.

With the help of the `ccpAdd` macro provided by Cocos2D-X, this calculates the summary of two points. Every time the `transformBoxes` function would be called it would update the origin point of the bounding boxes accordingly as demonstrated in Snippet 10.

```
_attackBox.actual.origin = ccpAdd (this->getPosition (), ccp
(_attackBox.original.origin.x, _attackBox.original.origin.y));
```

Snippet 10. Setting the attack's bounding box, doesn't update based on position.

This way the attack box will move with the actor sprite. The next problem arose when the player's character turned around facing to left and tried to attack an enemy. The attack box was still being created as if the player would be attacking to the right as can be seen in Figure 24.

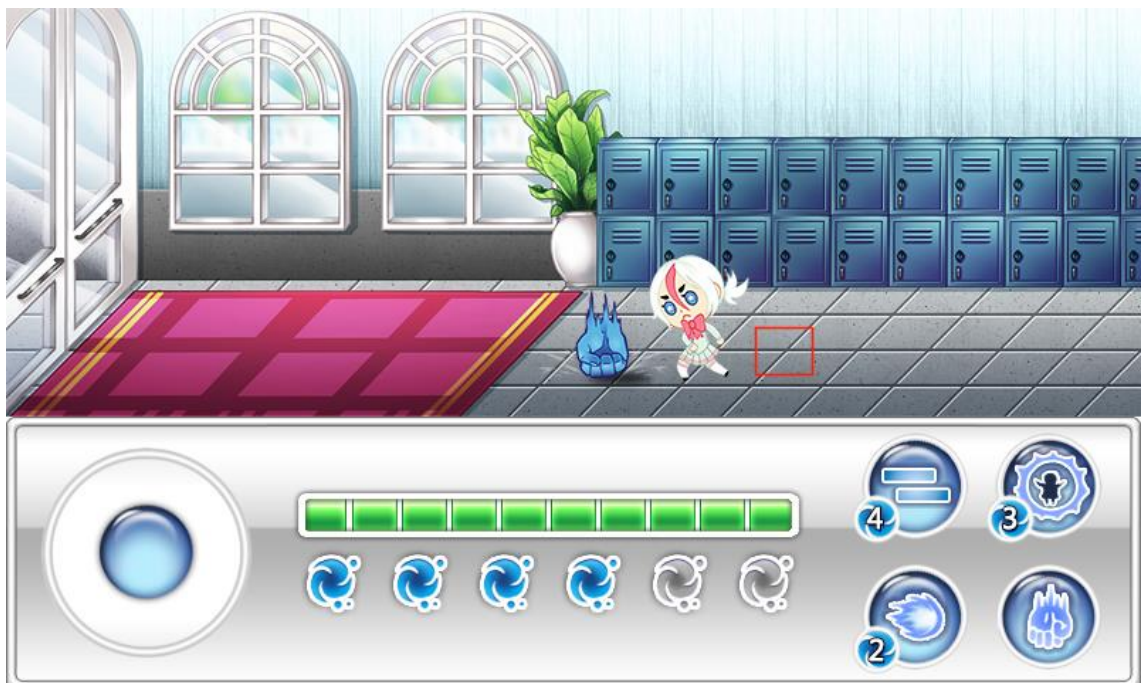


Figure 24. Showcasing the bounding box for the basic attack. The red border shows where the bounding box for the basic attack actually is.

The following solution was then developed for the aforementioned problem and can be seen in Snippet 11.

```
_attackBox.actual.origin = ccpAdd (this->getPosition (), ccp
(_attackBox.original.origin.x +
(this->getScaleX() == -1 ? (-_attackBox.original.size.width -
_hitBox.original.size.width): 0), _attackBox.original.origin.y));
```

Snippet 11. Setting the attack's bounding box, updates based on position.

A check has been added to check where the actor sprite is looking at, and based on that transforms the attack boxes origin point to create a forward thrusting attack.

### 3.5.3 Enemies

There was no complication in creating the player's character because it is a single object, but enemies had to vary in quantity. In an early iteration of the game enemies were created manually for testing purposes, but keeping up with a lot of enemies started to become very difficult and messy. The proposed solution was to create an array that holds reference to all the enemy objects and based on that could be accessed. The array was created using CArray object, which can hold different CCNodes. Using the CC\_SYNTHESIZE macro, which creates getters and setters for the desired object, in this case for the aforementioned enemy array. CC\_SYNTHESIZE macro takes three parameters, those being data type, variable name and method names respectively. This way enemy numbers could easily be changed, and going through all enemies was achieved by looping through the array.

### 3.5.4 Game AI

After researching popular brawler games, the following behavior of the enemy AI was noticed. The AI was very basic for the grunt type enemies, while becoming more complex for the bosses. In some games the bosses didn't have AI, but followed attack patterns that repeat themselves regardless of the situation. After reading about decision tree's in Artificial Intelligence for Games by Ian Millington, it was decided that decision trees would represent the enemies and boss AI. A decision tree is algorithm that uses a binary tree-like structure to display decisions and their possible results depending on event outcomes [29].

The current structure of the AI for the enemy grunts is as can be seen in Figure 25.

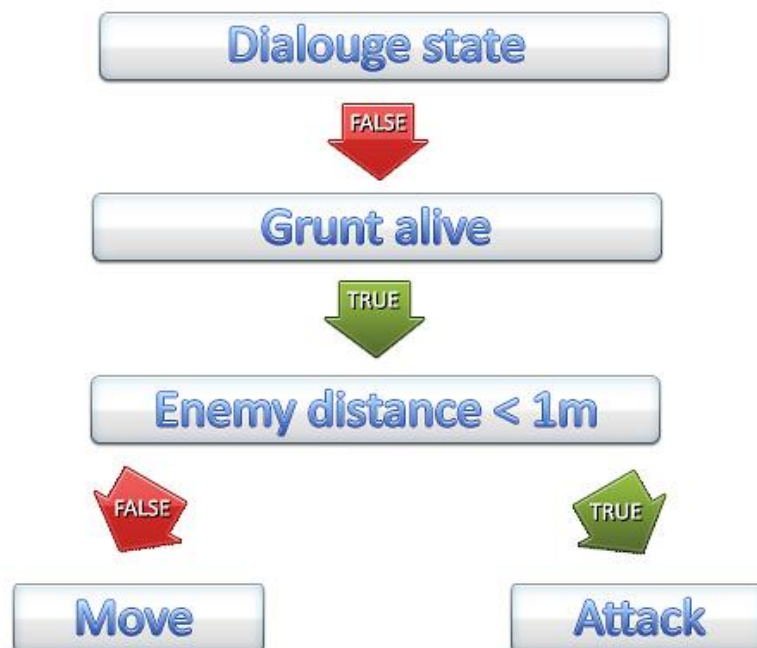


Figure 25. Grunt enemy decision tree.

As can be seen in Figure 25, the AI for the enemy grunts was kept compact. It follows the same pattern as other popular brawlers. If the enemy is not dead, and the player is within attack range, then attack or idle, else the enemy moves towards the player.

The boss enemy in this game has similar AI to the enemy grunts with a couple of tweaks as can be noted in Figure 26.

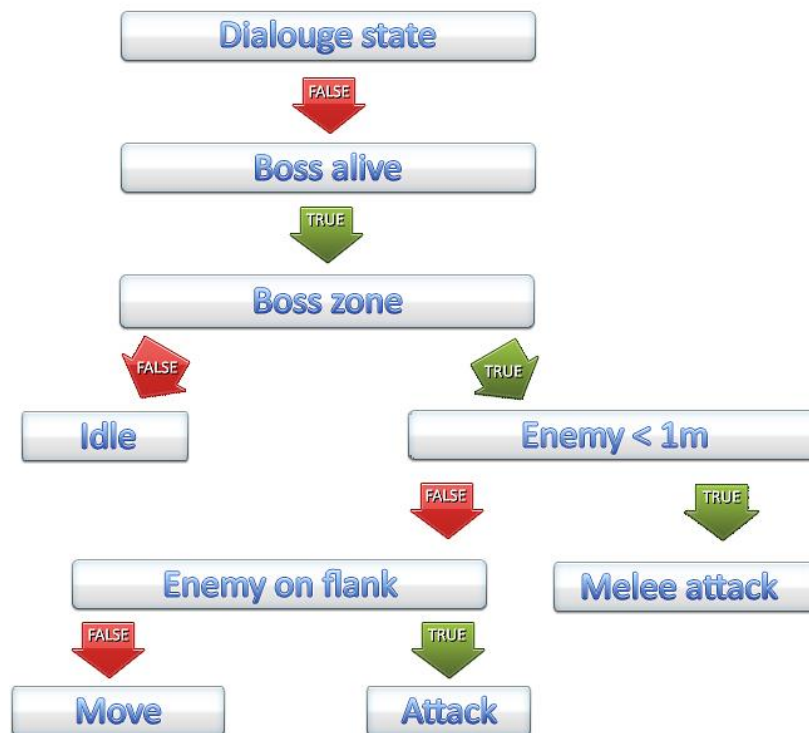


Figure 26. Boss enemy decision tree.

Instead of moving towards the enemy, the boss will position itself to the same horizontal plane as the player and shoot projectiles. The boss has two attacks to choose from at random, one that follows the player while the other would fly straight forward to the direction of the player. And if the player decides to stay too close to the boss, the player will get heavily damaged by the bosses close range melee attack.

### 3.5.5 Collision Detection

Collision detection is handled in the game layer by checking if the bounding boxes of the nodes intersect. In one of the very early versions of the game the checks for collisions were called very often in situations where collisions could not occur. That was a mistake, which caused performance problems when a lot of nodes were visible on the screen at the same time. The following solution was found, as can be seen in Snippet 12.

```

CObject *pObject = NULL;
CCARRAY_FOREACH(_enemies, pObject)
{
    EnemyFemale *enemy = (EnemyFemale*) pObject;

```

```

        if (enemy->getActionState() != kActionStateKnockedOut)
        {
            if (fabsf(_cherry->getPosition().y - enemy->getPosition().y) <
20)
            {
                if (_cherry->getAttackBox().actual.intersectsRect(enemy-
>getHitbox().actual))
                {
                    enemy->hurtWithDamage(_cherry->getDamage());
                    _cherry->setManaPool(_cherry->getManaPool() + 0.5f);
                }
            }
        }
    }
}

```

Snippet 12. Hit detection when the player hits enemy grunt(s).

Instead of being called all the time, the collision detection is only called when needed based on the state of the node. For projectiles attacks a similar solution was developed as can be seen in Snippet 13.

```

CCObject* it = NULL;
// for (it = _projectiles->begin(); it != _projectiles->end(); it++)
CCARRAY_FOREACH(_projectiles, it)
{
    CCSprite *projectile = dynamic_cast<CCSprite*> (it);
    CCRect projectileRect = CCRectMake (
        projectile->getPosition().x - (projectile->getContentSize().width/2),
        projectile->getPosition().y - (projectile->getContentSize().height/2),
        projectile->getContentSize().width, projectile-
>getContentSize().height);

    CCObject *pObject = NULL;
    CCARRAY_FOREACH(_enemies, pObject)
    {
        EnemyFemale *enemy = (EnemyFemale*) pObject;
        if (enemy->getActionState() != kActionStateKnockedOut)
        {
            if (projectileRect.intersectsRect(enemy->getHitbox().actual))
            {
                enemy->hurtWithDamage(_cherry->getProjectileDamage());
                projectile-
>runAction(CCSequence::create(CCCallFuncN::create(this, call-
funcN_selector(GameLayer::objectRemoval)), NULL));
            }
        }
    }
}

```

Snippet 13. Hit detection for projectiles.

As they are objects that move their bounding boxes had to be created dynamically and collision checks are made based on how many projectiles are active.



### 3.6 Game Performance

This chapter briefly covers game performance issues faced during the development of Project Cherry Brawl and solutions for said problems.

In 2D games there is mainly one thing that consumes most of the memory allocated to the game, and that is textures. Textures usually take 80% or more of the games' reserved memory [30]. That is why it is recommended to use sprite sheets and having optimized textures in general. Texture optimization means that for example a button texture should not have to be 520x520 pixels and then resized to 52x52px, but instead having the button's texture 52x52 pixels to begin with. This saves space and improves runtime performance.

When running an early prototype of Project Cherry Brawl, it was noticed that the game would slow down over the course of the gameplay. After a couple of build iterations, it was noticed that the game was using too many OpenGL draw calls. By using the `CCSpriteBatchNode` the games' draw call count was greatly reduced.

The second issue appeared when the player or enemy would shoot a projectile and it would go out of the player's field of view. The projectile was kept being displayed off screen, which in the long run caused the game to slow down. The issue was fixed by removing projectiles after a certain distance was covered, if they did not collide with other `CCNodes`.

The third problem was creating damage numbers after an attack had been made on an enemy or the player's character as can be seen in Figure 27.

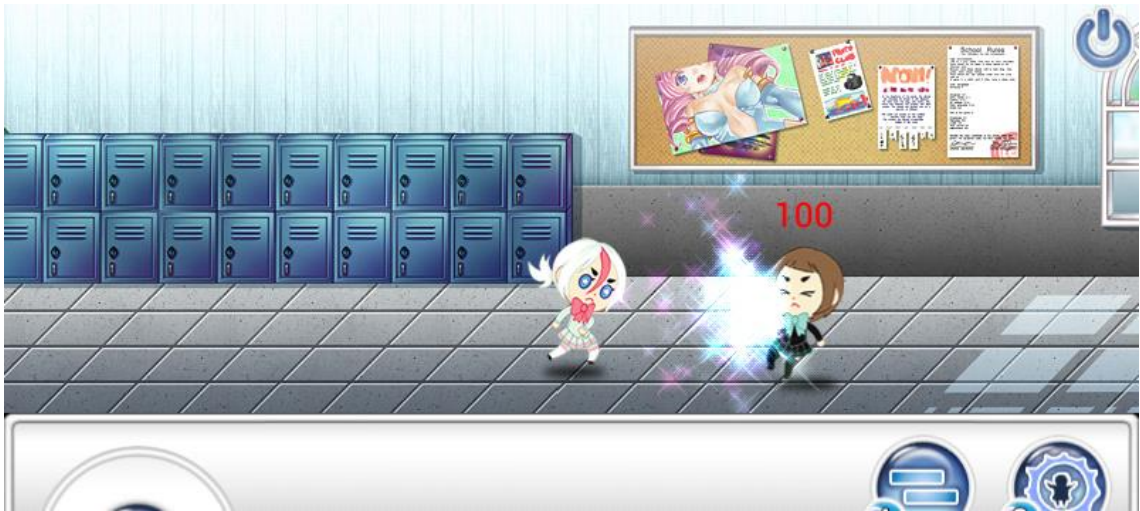


Figure 27. Damage numbers, taken from Project Cherry Brawl.

The problem there was that the number sprite was being hidden with the `CCFadeOut` action. The `CCFadeOut` makes a `CCNode` fade out after certain duration. The issue with that was that it was being still displayed, but with an opacity of 0, wasting an OpenGL draw call. The solution for this problem was to add the `CCHide` action into the sequence of actions. `CCHide` hides a `CCNode`, thus avoiding an additional OpenGL draw call.

## 4 Discussion and Conclusions

In this thesis Linda Karlsson and the author created a game from the very beginning to a state where it could be put on the Google Play Store for people to try it out. During the process of this project a lot of difficulties were faced because this was something neither had worked on before and therefore had no experience with.

When the development of Project Cherry Brawl started Cocos2D-X v2.2.3 was the most stable version of Cocos2D-X. Two months into game's development the Cocos2D-X team released v3.0 of Cocos2D-X and as to the updated features [31], it was a major update to the previous version. Unfortunately due to time constraints with this project it was necessary to stick with v2.2.3.

While working on Project Cherry Brawl there were random crashes. At the time Eclipse was being used to as the main IDE but the error stack trace that it was giving was unreadable and the only interpretation was that it had something to do with memory allo-

cation. This was the reason it was chosen to continue the development with Visual Studio 2012. With Visual Studio it was possible to debug the project and get rid of the random crash occurrences. Cocos2D-X provided insight on how game engines work and their workflow. This information can be carried over into other game engines such as Unity or LibGDX.

Project Cherry Brawl was a valuable experience throughout the process of its creation, but things could have been done differently. It tried to emulate the feeling of having a controller and playing a console game on the phone, which has been proven many times that it, seemingly did not work very well on mobile devices. The phones touch and gesture features were not taken advantage of but that was personal preference of the creators because it was intended to emulate the old school style of playing games with a controller.

It was unsure whether or not Project Cherry Brawl should be continued after its release on several Android Markets, such as Google Play Store and SlideMe [32], but after having the game on there for a week, our team received a lot of positive feedback. At first the brawler genre was taught to only work on consoles or PC's, but it became apparent that there is demand for the genre on mobile platforms as well. It would work very well on PC's, because there is a keyboard and mouse to work with. Thus the developers of Project Cherry Brawl decided to continue with the development for the Android platform and eventually expand to other mobile devices such as iOS and Windows Phone. Additionally, with the recently released Cocos2D-X 3.0 and its support for C++11 and OpenGL 2.0, the game will perform better on lower-end devices and provide a smoother experience for the players.

## References

- 1 [http://yle.fi/uutiset/suurmenestys\\_supercellin\\_toimitusjohtaja\\_kaikki\\_lahtee\\_tekijatiimista/6584310](http://yle.fi/uutiset/suurmenestys_supercellin_toimitusjohtaja_kaikki_lahtee_tekijatiimista/6584310) , 1.1.2014.
- 2 <http://www.cocos2d-x.org/> , 5.1.2014.
- 3 <http://www.cocos2d-x.org/wiki/Cocos2d-x> , 7.1.2014.
- 4 [http://www.cocos2d-x.org/wiki/Engine\\_Architecture](http://www.cocos2d-x.org/wiki/Engine_Architecture), 3.2.2014.
- 5 [http://www.cocos2d-x.org/reference/native-cpp/V2.2/d1/da4/classcocos2d\\_1\\_1\\_c\\_c\\_scene.html#ae3c0dadfbfae64c9dd1d5d9a6bec3d42](http://www.cocos2d-x.org/reference/native-cpp/V2.2/d1/da4/classcocos2d_1_1_c_c_scene.html#ae3c0dadfbfae64c9dd1d5d9a6bec3d42), 3.3.2014.
- 6 <http://www.mapeditor.org/>, 4.3.2014.
- 7 <https://github.com/bjorn/tiler/wiki/TMX-Map-Format>, 15.4.2014.
- 8 <http://www.codeandweb.com/texturepacker>, 15.4.2014.
- 9 <http://www.cocos2d-x.org/docs/tutorial/parkour-game-with-cocostudio/chapter1/en>, 2.2.2014.
- 10 <http://libgdx.badlogicgames.com/>, 15.4.2014.
- 11 <https://unity3d.com/>, 15.4.2014.
- 12 <http://unity3d.com/pages/2d-power?gclid=CLvjgp-j470CFasLcwodUp0A9g>, 15.4.2014.
- 13 <http://unity3d.com/unity/whats-new/unity-4.3>, 15.4.2014.
- 14 <http://www.learn-cocos2d.com/2013/11/mobile-game-engine-popularity-index/>, 15.4.2014.
- 15 <https://play.google.com/store/apps/details?id=com.letang.game103pp.cn>, 22.3.2014.
- 16 [http://www.gamasutra.com/view/feature/132674/game\\_ui\\_discoveries\\_what\\_players\\_php](http://www.gamasutra.com/view/feature/132674/game_ui_discoveries_what_players_php), 16.1.2014
- 17 [http://www.cocos2d-x.org/wiki/Multi\\_resolution\\_support](http://www.cocos2d-x.org/wiki/Multi_resolution_support), 3.1.2014.

- 18 [http://en.wikipedia.org/wiki/Double\\_Dragon](http://en.wikipedia.org/wiki/Double_Dragon), 15.4.2014.
- 19 [http://en.wikipedia.org/wiki/Streets\\_of\\_Rage](http://en.wikipedia.org/wiki/Streets_of_Rage), 15.4.2014.
- 20 [http://en.wikipedia.org/wiki/Final\\_Fight](http://en.wikipedia.org/wiki/Final_Fight), 15.4.2014.
- 21 [http://en.wikipedia.org/wiki/Teenage\\_Mutant\\_Ninja\\_Turtles:\\_Turtles\\_in\\_Time](http://en.wikipedia.org/wiki/Teenage_Mutant_Ninja_Turtles:_Turtles_in_Time), 15.4.2014.
- 22 <http://igda.fi/>, 15.4.2014.
- 23 <http://igda.fi/?p=4109>, 15.4.2014.
- 24 <http://dena.com/intl/>, 15.4.2014.
- 25 <http://www.cocos2d-x.org/forums/6/topics/16774>, 4.3.2014.
- 26 <http://www.cocos2d-x.org/wiki/Cocos2d-JS>, 12.4.2014.
- 27 <http://en.wikipedia.org/wiki/D-pad>, 15.4.2014.
- 28 <http://entropyinteractive.com/2011/02/game-engine-design-the-game-loop/>, 16.4.2014.
- 29 Artificial Intelligence for Games by Ian Millington. Chapter 5.2. Page 303.
- 30 [http://www.cocos2d-x.org/wiki/How\\_to\\_Optimise\\_Memory\\_Usage](http://www.cocos2d-x.org/wiki/How_to_Optimise_Memory_Usage), 27.3.2014
- 31 [https://github.com/cocos2d/cocos2d-x/blob/develop/docs/RELEASE\\_NOTES.md](https://github.com/cocos2d/cocos2d-x/blob/develop/docs/RELEASE_NOTES.md), 20.4.2014
- 32 <http://slideme.org/> 1.5.2014.